A line follower robot implementation using Lego’s Mindstorms Kit and Q-Learning

Victor Ricardo Cruz-Álvarez*, Enrique Hidalgo-Peña* and Hector-Gabriel Acosta-Mesa*

**ABSTRACT**

A common problem working with mobile robots is that programming phase could be a long, expensive and heavy process for programmers. The reinforcement learning algorithms offer one of the most general frameworks in learning subjects. This work presents an approach using the Q-Learning algorithm on a Lego robot in order for it to learn “by itself” how to follow a black line drawn down on a white surface, using Matlab [5] as programming environment.

**INTRODUCTION**

Programming robots in a traditional way could be a very large process, in which programmers have to map from sensor to actuators in many iterations. Robot sensors and actuators are not human-like: misconception about how they operate can cause control fails. Another approach is learning from experience and creating appropriate adaptive control systems.

A general approach is the framework provided by reinforcement learning, which requires an unambiguous representation of states and actions and the existence of a scalar reward function. This means, that if robot takes an action, it’ll drive the robot from a “current” state to a “final” state, depending on the action taken. So, each action is rewarded or punished and, this way, the robot “learns” how to accomplish a specific task, and the programmer is just worried about other subjects like sensor calibration, communication or the environment dising, among others.

**Reinforcement Learning**

Reinforcement learning (RL) is a machine learning paradigm [1][2] that is particularly well-situated for using on mobile robots [6][7][8][9]. This kind of learning can be described as “a way for programming agents that learn, by reward and punishment, without the need of specifying how to accomplish the task”.

The RL assumes that the world can be described by a set of states $S$, and that the agent (the robot, in this case) can take one from a finite number of actions $A$. The time is divided in discrete steps, and for each step, the robot observes the state of the world, $S$, and chooses an action $a$. After taking the action, the reward function (RF) gives a reward $r$ to the robot.
The model of reinforcement learning is shown next:

1. Check the state $S_t$
2. Decide an action $a_t$
3. Do that action,
4. Check the new state $S_{t+1}$
5. Get the reward $r_{t+1}$
6. Learn from experience (relate state-action reward value and store it).
7. Repeat.

The fact of specifying what to do, but not how to do it, allows better final solutions, because they are based on the agent’s real experience and not on programmer’s assumptions. The advantage of this is that programmer’s assumptions could be ambiguous, or hard to mathematically model them.

The RL is based on Markovian Decision Process (MDP)[3], as is briefly described in the next subsection.

**Markovian Decision Process**

Formally, it is:

- A set of states, $S = \{s_1, s_2, ..., s_n\}$
- A set of actions, $A = \{a_1, a_2, ..., a_n\}$
- A reward function, $R : S \times A \times S \rightarrow \mathbb{R}$
- A transfer function, $P^{i}_{ij} = P(s_{t+1} = j \mid s_t = i, a_t = a)$

Reinforcement Learning is a particular case of MDP, in which the sets of states and actions are known, but the transfer or the reward function is missed. This is, Markovian Decision Process is incomplete.

**Q-Learning**

The particular reinforcement learning algorithm that we use in this work is Q-Learning [4]. This algorithm approximates the value of state-action function $Q$ iterating the process. For learning with this algorithm, we keep an estimate of the $Q(s, a)$ function in a table. These values are updated as the agent achieves more experience. The estimate of $Q$ doesn’t depends on exploration. The Q-Learning algorithm pseudocode is shown in the following subsection.

**Algorithm**

1. Initialize $Q(s, a)$ on small random values, $\forall s, \forall a$
2. Check the state $s$
3. Pick an action $a$ and execute it out.
4. Check new state $s'$ and reward the last action.
5. $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$
6. Go to step 2.

Where:

- $Q(s, a)$ is the learning function,
- $s \in S$ is the current state,
- $a \in A$ is the executed action,
- $s' \in S$ is the state driven by the action $a$
- $0 < \alpha$ is the learning rate, and
- $0 \leq \gamma$ is the discount rate

The content of this paper is divided as follows: Section Experiments shows the implementation, how the Lego was connected with Matlab, the elements used to build the prototype, the states and actions set used for the learning process and the parameters for the execution of the algorithm. In Section Results and Discussion, the results of the experiment are discussed, and the final set of states and actions are shown, and also the reward values plots given by the algorithm. Finally, conclusions and future work are presented in Section Conclusions and Future Work.

**EXPERIMENTS**

For this experiment, we used Lego Mindstorms Kit. Also, we programmed the algorithm in Matlab, using the RWTH 4.03 Library. In this section, we discuss how we accomplish this phase of the experiment.

**Communicating NTX with Matlab**

On principle, we analyze the possibility of interconnecting Matlab with the NXT using a Bluetooth link. The RWTH brings this communication using a USB Bluetooth dongle. Alternatively, it is possible to use a USB cable to connect with the NXT.

On the first communication tests, we discover that NXT allows communication via a Bluetooth link, but when a test program ends, the communication is interrupted unless Bluetooth service is rebooted in the NXT.
Considering this fact, we thought that we can be in front of an unstable connection, and in order to minimize the risk of lose communication, we decided to use the USB connection via cable.

The initial proposal of using Bluetooth dongle was done in order to avoid the USB cable could interfere in the learning phase of the robot, and with this give to robot the freedom of movement since is necessary the robot to wander around for the algorithm to work properly.

Prototype building

The following Lego’s Mindstorms Kit elements were used in order to build up our prototype:

- NXT 2.0 IntelliBrick, firmware 1.29.
- 2 optic sensors (active mode) on the input ports 2 and 3.
- 2 servos for the wheels on output ports B and C.
- 1 ultrasonic sensor for avoiding obstacles / walls on input port 4.

States and actions

Initially, the prototype doesn’t know how to perform its task. So, to follow the line (an amoeba-shaped closed curve), the prototype has to wander around on the test ground space, until the algorithm builds a weight table that is going to be used in another program to follow the line successfully.

First, we defined four states and three actions, as follows:

- $S_1$: Both sensors detect the line.
- $S_2$: Left sensor doesn’t detect line.
- $S_3$: Right sensor doesn’t detect line.
- $S_4$: None of both sensors detect line.

- $a_1$: Go forwards.
- $a_2$: Turn to the left.
- $a_3$: Turn to the right.

Also, we build an state graph, in which, obviously, any action in any state takes to another one, including the same from which the action is performed. This is because initially the robot doesn’t know how to follow the line, and it will try to learn how to do it, according to the algorithm.

![Initial State graph.](image)

Originally, we proposed a fifth state, given by the ultrasonic sensor, but in order to reduce the model’s complexity, we decided to make a simple reactive behaviour for avoiding obstacles and walls, in which, the so calibrated sensor for 20 centimeters, when detecting something, it makes a backwards movement, and then it turns 45 degrees to the right. This helps getting back the robot on the wandering around the ground and, eventually, back on detecting line.

The initial reward was set to a value of 100, but when executing the first tests, we noted this value was generating high values of $\gamma$, and in order to avoid big integers ($> 1 \times 10^{18}$), we decided to set the reward on 10. It is worth to mention that, as appreciated in figure 2, the state $(S_4, a_1)$, has a reward value of zero. This is for avoiding the robot to get stucked in this state, and helps it saving time in the learning process.

Algorithm execution

Once upon prototype was build, the algorithm was executed from Matlab. The tests were repeated for $\alpha$ values of 0.5 and 0.8 for comparison purposes.

The Q-Learning program was executed once for each value of $\alpha$. Each one of the executions has a cycle of
2400 iterations, taking an mean time of 2 hours for each test. The results are discussed in the next section.

RESULTS DISCUSSION

In order to register the time evolution of the algorithm, we paused the execution at every 200 iterations. On each pause, we registered down the variable that stored the weight table. This data were subsequently dumped into an spreadsheet for further analysis and plotting purposes. When the algorithm execution reached the 2400 iterations, the results given by it allowed us to test the robot’s “learned” behavior. At this point is possible to say that Q-Learning has allowed the robot to learn how to follow the line. The plots and state graphs placed next show this is valid, since it’s appreciated in the plots in figures 3 and 6 that the three highest values corresponds to the state graphs in figures 5 and 8 respectively, and they are specifically the actions that keep the robot reading the line once it has learned in both cases.

Results for $\alpha = 0.5$

It is possible to appreciate in figure 3 the combination of states-actions in horizontal axis, and the final value of the rewards, in vertical axis. The ID shown in the horizontal axis of the plot matches with a combination of an initial state, an action taken, and the state driven by that action. This set of combinations of states and actions for $\alpha = 0.5$ is shown in the table in figure 4. The states and actions labels where previously discussed in section Experiments (Prototype Building).

As a consequence of this results, we were able to build a new state graph, shown in figure 5, where is appreciated that the state $S_4$ from fig. 2, at which the robot doesn’t reads the line, no longer exists (as we mentioned previously at the begining of this section), which means that at this point, the robot is following the line in every action it takes. This could be understood as the robot had learned from its own experience, this is, of course, because of Q-Learning results.

<table>
<thead>
<tr>
<th>ID</th>
<th>Detecting</th>
<th>Action</th>
<th>Detecting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BothSensors</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>2</td>
<td>BothSensors</td>
<td>Forward</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>3</td>
<td>BothSensors</td>
<td>Forward</td>
<td>RightSensor</td>
</tr>
<tr>
<td>4</td>
<td>BothSensors</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>5</td>
<td>BothSensors</td>
<td>Left</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>6</td>
<td>BothSensors</td>
<td>Right</td>
<td>BothSensors</td>
</tr>
<tr>
<td>7</td>
<td>BothSensors</td>
<td>Right</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>8</td>
<td>BothSensors</td>
<td>Right</td>
<td>RightSensor</td>
</tr>
<tr>
<td>9</td>
<td>LeftSensor</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>10</td>
<td>LeftSensor</td>
<td>Forward</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>11</td>
<td>LeftSensor</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>12</td>
<td>LeftSensor</td>
<td>Left</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>13</td>
<td>LeftSensor</td>
<td>Right</td>
<td>BothSensors</td>
</tr>
<tr>
<td>14</td>
<td>LeftSensor</td>
<td>Right</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>15</td>
<td>RightSensor</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>16</td>
<td>RightSensor</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>17</td>
<td>NoneSensor</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>18</td>
<td>NoneSensor</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>19</td>
<td>NoneSensor</td>
<td>Right</td>
<td>BothSensors</td>
</tr>
</tbody>
</table>

Results for $\alpha = 0.8$

In the same way, we present the result of the Q-Learning execution with $\alpha = 0.8$. The plot is shown in figure 6:
The vertical axis is shown in logarithmic scale, because the high values of combinations 4, 8 and 11 don’t let to appreciate with clarity the rest of the combinations. This combinations are presented in the table 7.

Table 2.

<table>
<thead>
<tr>
<th>ID</th>
<th>Detecting</th>
<th>Action</th>
<th>Detecting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BothSensors</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>2</td>
<td>BothSensors</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>3</td>
<td>BothSensors</td>
<td>Right</td>
<td>BothSensors</td>
</tr>
<tr>
<td>4</td>
<td>BothSensors</td>
<td>Right</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>5</td>
<td>LeftSensor</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>6</td>
<td>LeftSensor</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>7</td>
<td>LeftSensor</td>
<td>Right</td>
<td>BothSensors</td>
</tr>
<tr>
<td>8</td>
<td>LeftSensor</td>
<td>Left</td>
<td>RightSensor</td>
</tr>
<tr>
<td>9</td>
<td>RightSensor</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>10</td>
<td>RightSensor</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>11</td>
<td>RightSensor</td>
<td>Left</td>
<td>LeftSensor</td>
</tr>
<tr>
<td>12</td>
<td>NoneSensors</td>
<td>Forward</td>
<td>BothSensors</td>
</tr>
<tr>
<td>13</td>
<td>NoneSensors</td>
<td>Left</td>
<td>BothSensors</td>
</tr>
<tr>
<td>14</td>
<td>NoneSensors</td>
<td>Right</td>
<td>BothSensors</td>
</tr>
</tbody>
</table>

With this values, it were also possible to build a state graph shown in figure 8, where, as in the case of $\alpha = 0.5$, state $S_4$ no longer exists, and the robot follows line regardless the action taken. Again, the robot had learned from its own experience.

It is worth to mention that the results show that, although the main goal was achieved successfully in both cases, the specific way to make it was not the same. For the case of $\alpha = 0.5$, the plot and state graph tell that robot learned to follow the line with the stimulus of the right sensor’s readings, while in the case of $\alpha = 0.8$ the learning was possible mainly by the left sensor’s readings. This is because the initial wandering, and, of course, since this wandering is done randomly.

**CONCLUSIONS AND FUTURE WORK**

Once we collect the data shown above, we proceed to write another program that uses the weight tables obtained from the former tests in order to control the robot. The execution of this new program allowed us to verify that the Q-Learning algorithm actually makes the robot “to learn” how to accomplish a task by itself. After watching it wander around randomly on the test ground, when the second program controls the robot, it follows the line without troubles, and with this fact, we can validate the data given by the results and the state graphs from the executions, so this algorithm becomes very useful to reduce the complexity of programming for robot’s tasks, since the learning process relies on Q-Learning Algorithm, and not on programmer’s skills or assumptions.

For future work another Artificial Intelligence techniques and algorithms are proposed to be used for training the robot’s behavior, for example: genetic algorithms[10][11], cultural algorithms[12], artificial neural networks and differential evolution[13] or even a combination of them could be possible. This way, depending of the robot’s task, one technique or another could be choosen in order to achieve optimum learning values and the optimal tuning of the behavior. An special improvement to this work using this techniques is the sensor calibration stage. This way, it could be possible to get a more accurate sensor thresholds, and even robot would be capable of recalibrating sensors for different light conditions. This fact is of interest since one of the hard tasks configuring mobile robots for natural enivroments is the light sensor calibration.
REFERENCES


